Automatically Selecting and Using Primary Effects in Planning: Theory and Experiments

Eugene Fink and Qiang Yang

October 1994
CMU-CS-94-206

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

The first author is supported by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. The second author is supported in part by a grant from the Natural Science and Engineering Research Council of Canada. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Wright Laboratory or the U. S. Government. The U. S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation thereon. This manuscript is submitted for publication with the understanding that the U. S. Government is authorized to reproduce and distribute reprints for Governmental purposes.

Keywords: planning, problem solving, learning

Abstract

Using primary effects of operators in planning is an effective approach to reducing planning time and improving solution quality. However, the characterization of "good" primary effects has remained at an informal level. In addition, no method has previously been known to automatically learn the primary effects of operators from a given domain specification. In this paper we formalize the use of primary effects in planning, present a criterion for selecting useful primary effects that guarantee the efficiency and completeness of planning, and prove the near-optimality of solutions found by planning with primary effects. Based on the formalization, we describe an inductive learning algorithm that automatically selects primary effects of operators. We show that the learning algorithm performs efficiently, producing plans that are near-optimal with high probability. We also empirically demonstrate the effectiveness of the learned primary effects in reducing search.

1 Introduction

We are interested in automatically finding primary effects of planning operators to improve planning efficiency and solution quality. In plan construction, an operator might have a large number of effects, only a few of them being of special importance to the problem at hand. The idea of primary effects is to specify certain effects of a planning operator as *primary*, and to use an operator during planning only for the purpose of achieving its primary effects. A *primary-effect restricted* planner never inserts an operator into a plan for the sake of its side effects.

In this paper we formalize the intuition behind the use of primary effects and present a learning algorithm to automatically select primary effects with good quality. To begin with, we consider the following motivating examples.

1.1 Motivating Examples

Example 1.

Imagine a house with a fireplace in the living room. The fireplace may be used to warm and to illuminate the room. If the tenant of this house has electric lamps in the living room, she would probably view illuminating the room as a *side* effect of lighting the fireplace. That is, she would not use the fireplace if her *only* goal were to illuminate the room, since electric lamps are easier to use and electricity is cheaper than wood. Thus, warming the room is a primary effect of using the fireplace.

Example 2.

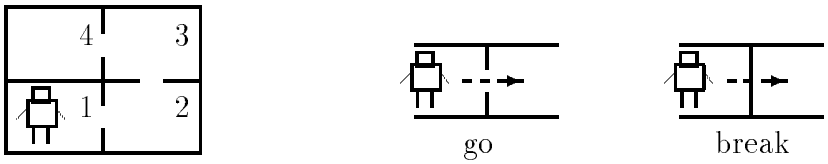
In our second example, suppose that you are going to a computer store to buy diskettes. The primary effect of this action is obtaining diskettes — this is your main goal. Side effects are spending \$20, having a pleasant walk on a sunny day, and so on.

We consider this example in a little more detail. One would not go to a computer store *in order* to spent \$20 or have a pleasant walk, because these goals could be achieved more easily by some other actions. For example, if your only goal were to have a walk, you would probably go to a nearby park rather than to a computer store. Thus, one aspect of using primary effects is to view a result of an action as a side effect if this result can always be achieved by another, cheaper plan.

In this example the quality of a solution plan is defined as the sum of the costs of operators used in the plan. In this model, the smaller the total cost of the operators, the better the quality of the plan.

Example 3.

Our next example is a modified version of the robot world from [Fikes and Nilsson, 1971]. This world consists of four rooms and a robot (see Figure 1a). The robot can perform two different actions: **go** between two rooms connected by a door and **break** through the wall to create a new door (see Figure 1b). The **break** operator not only creates a new door, but also moves a robot to the room behind the broken wall.



(a) A "map" of the robot world (b) Operators

Figure 1: The robot world.

Notice that changing the location of a robot in this world can always be accomplished by a series of **go** operators, without breaking through walls. Thus, the change of robot's location can be considered as a side effect of the **break** operator. When a planner searches for a plan for moving the robot, it may use this knowledge to reject the **break** alternative and consider only the **go** operators. We show in Section 4 that this saving in the number of considered alternatives grows exponentially with the search depth. Thus, reducing the number of alternative choices of operators during planning could improve the search efficiency.

1.2 Overview

The advantage of using primary effects has been recognized early in planning research. STRIPS [Fikes and Nilsson, 1971] utilized primary effects as a means to control the quality of solution plans. SIPE [Wilkins, 1988] distinguished between the "main effects" and the "side effects" of operators and used this distinction to simplify conflict resolution in planning. PRODIGY [Carbonell *et al.*, 1991, Knoblock, 1991a] allows the user to specify primary effects of operators, which are then used as an effective method to control search. The ABTWEAK planner [Yang and Tenenberg, 1990, Yang, Tenenberg and Woods, 1991] also allows the user to specify primary effects, the use of which significantly improves the efficiency of planning.

Despite the importance of primary effects, the characterization of a "good" selection of primary effects has remained on an informal level. Furthermore, no method has been proposed to automatically select primary effects for a given domain definition. All existing systems rely on the human user to select primary effects. If the user has not chosen primary effects, then by default all effects are assumed to be primary.

The lack of formal guidelines for selecting primary effects could cause three serious problems in planning. First, an improper selection of primary effects may result in the loss of completeness in planning. This happens when a planner that uses primary effects cannot find a plan for a solvable planning problem. In Example 1, if a fireplace is the only source of light, but lighting is not chosen as a primary effect of using fireplace, a primary-effect restricted planner will not find a plan for illuminating the room.

Second, ill-selected primary effects may produce non-optimal solutions to planning problems. This could happen because primary effects place a bias in directing the search for a solution. If not set properly, the bias can favor a search path toward a costly solution. In Example 3, if the change of location is taken to be one of the primary effects of the *break*

operator but not of the *go* operator, then a planner would only consider breaking walls as a means for the traveling from one room to another. The cost of the resulting solutions may be much higher than simply going between rooms.

Third, although using primary effects can reduce the branching factor of search, the search depth could increase if the primary effects are not chosen correctly. With a best-first search, the search depth is determined by the cost of the optimal solution to a problem. However, an ill-selected set of primary effects can preclude the optimal solution from even being present in the search space of a planner. With a more costly solution plan, the search depth could be dramatically increased, leading to a less efficient planner than one that uses all operator effects.

The purpose of this paper is twofold. First, we pinpoint the exact reason for a primary-effect restricted planner to be incomplete and non-optimal. This result is presented in the form of a theorem, which states the *necessary and sufficient* condition for a primary-effect restricted planner to be complete and to produce solutions that are good approximations to the optimal solution. Thus, the theorem formalizes the intuitions behind "good" choices of primary effects. Using the theorem, it is now possible to determine whether planning with primary effects preserves completeness and good solution quality.

Second, we present an inductive learning algorithm to automatically select primary effects. Although learning the best set of primary effects in general is undecidable, with our learning algorithm we guarantee that the learned primary effects exhibit the following properties:

(a) The completeness of planning is preserved with high probability.
(b) The planner almost always finds a near-optimal solution.
(c) The use of primary effects dramatically reduces the planning time.

An initial set of primary effects can be first chosen by the user or by an initialization algorithm. The learner then uses either a collection of previous plans or new plans generated by the planner as examples to augment the set of primary effects of operators. Our analysis shows that the use of the learned primary effects considerably reduces the planning time. Furthermore, through empirical tests, we show that the use of the learned primary effects leads to dramatic improvement of planning efficiency.

1.3 Basic Assumptions

Search complexity and solution quality are two main factors that concern most classical planners. In this paper, we assume that we are working with a planning algorithm based on best-first search. We also assume that every operator has a numerical cost value, and the cost of a solution plan is the sum of the costs of operators in the plan.

Under these assumptions, a best-first-search based planner should be able to find the optimal solution when no distinction is made between primary and side effects. However, without using primary effects the search space might be too large to be handled by the planner. When primary effects are used, the branching factor of search would decrease, while both the search depth and the solution cost might increase. Our basic approach then is to strike a good balance between the branching factor and the search depth so that when

combined they result in a more efficient search than without using primary effects. At the same time, we make sure that the cost of a solution is within a small constant factor from the cost of the optimal solution.

In our discussion and implementation we use a version of the TWEAK planner [Chapman, 1987]. However, the theory of primary effects and the learning techniques presented in the paper are not limited to TWEAK. The only requirement is that during the learning phase a planner that is capable of finding an optimal solution to planning problems is used.

Historically, primary effects were also used to enable depth-first search planners, such as STRIPS, to improve the solution quality by directing search to low-cost solution plans. In this article we do not address this use of primary effects and concentrate on the problem of reducing planning time.

To summarize, the main purpose of this article is to improve the efficiency of classical planning algorithms by using primary effects. Primary effects are selected by a learning algorithm, whose goal is to minimize the planning time while ensuring a near-optimal quality of the solutions generated by the planner, where the quality of a solution is defined as the total cost of its operators.

1.4 Outline of the Article

The article is organized as follows. First, we formalize the notion of planning with primary effects (Section 2) and describe conditions that ensure the completeness of planning and solution quality (Section 3). Then we analyze the search space of planners that use primary effects and derive conditions under which the use of primary effects increases the efficiency of planning (Section 4). In Section 5 we present an inductive learning algorithm that automatically chooses primary effects, and in Section 6 we derive the time complexity and sample complexity of the algorithm. In Section 7 we present a series of experiments, demonstrating the effectiveness of the primary effects found by the learning algorithm in reducing planning time. Section 8 discusses possible extensions to the learning algorithm.

2 Using Primary Effects in Planning

In this section we introduce the basic concepts used throughout the paper, describe the use of primary effects in planning, and present the Prim-TWEAK planner, a version of TWEAK [Chapman, 1987] for planning with primary effects. First, we define planning domains of the TWEAK planner and plans generated by TWEAK (Section 2.1). Then we describe the use of primary effects in planning (Section 2.3) and formalize the notion of primary-effect justified plans (Section 2.2).

2.1 Definition of a Planning Domain

A *planning domain* is defined by a library of operators. Each *operator* α is defined by a set of *precondition literals* $Pre(\alpha)$ and a set of *effect literals* $Eff(\alpha)$. If a literal l is an effect of α, we say that α *achieves* l. If $\neg l$ is an effect of α, we say that α *deletes* l. A literal is *achievable* if it can be achieved by at least one operator in the library.

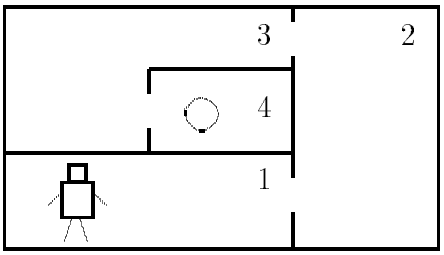


Figure 2: The robot planning domain.

A planning problem is defined by an *initial state* and a *goal state*, where a *state* is a set of literals. A *total-order plan* is a finite sequence of operators. We say that a precondition l of some operator α in a total-order plan is *satisfied*, if there is an operator α_1 before α that achieves l, and no operator between α_1 and α deletes l. For simplicity, we view an initial state as the first operator of a plan, denoted as α_{init}. This operator has effects that are identical to the initial-state literals and has no preconditions. Similarly, the last operator of a plan, α_{goal}, represents the goal of the plan. Its preconditions are identical to the goal literals, and its effect set is empty. A total-order plan is called *correct* if all preconditions of all operators are satisfied. In this case we say that the plan *solves* the planning problem $(\alpha_{\text{init}}, \alpha_{\text{goal}})$.

A *partial-order plan* Π is a partially ordered set of operators. The partial order represents the order of execution of the operators in the plan. As in total-order plans, α_{init} precedes all other operators, and α_{goal} is preceded by all other operators. A *linearization* of a partial-order plan is a total order of the operators consistent with the plan's partial order. We say that a statement about a partial-order plan is *possibly* true if it is true for *some* linearization of the plan. For example, an operator α_1 is possibly before α_2 if α_1 occurs before α_2 in one of the linearizations. We say that a statement is *necessarily* true if it is true for *all* linearizations of the plan. In particular, a partial-order plan is *necessarily correct* (or simply *correct*) if all its linearizations are correct.

The number of operators in a plan, *not* including the initial and goal states, is called the *size* (or *length*) of the plan and usually denoted by n.

Example We describe an extended version of the robot domain presented in Section 1.1 (see Figure 2). In our robot world, a ball is initially located in room 4. To describe a current state of the domain, we have to specify the locations of the robot and the ball, and the connectivity of rooms by doors. This may be done with three predicates, `robot-in`(x), `ball-in`(x), and `door`(x,y). Literals describing a current state of the domain may be obtained from these predicates by substituting specific room numbers for x and y. For example, the literal `robot-in`(1) means that the robot is in room 1, `ball-in`(4) means that the ball is in room 4, and `door`(1,2) means that room 1 and room 2 are connected by a doorway.

In this simple robot domain we assume that the doors are always open. The robot may go between two rooms connected by a door, break through a wall, carry the ball through a door, or throw the ball through a door into another room. These operators are described in

Operator	Preconditions	effects	cost
go(x,y)	`robot-in`(x), `door`(x,y)	`robot-in`(y), $\neg$`robot-in`(x)	2
carry-ball(x,y)	`robot-in`(x), `ball-in`(x) `door`(x,y)	`robot-in`(y), `ball-in`(y) $\neg$`robot-in`(x), $\neg$`ball-in`(x)	3
throw(x,y)	`robot-in`(x), `ball-in`(x) `door`(x,y)	`ball-in`(y), $\neg$`ball-in`(x)	2
break(x,y)	`robot-in`(x)	`robot-in`(y), $\neg$`robot-in`(x) `door`(x,y)	4

Table 1: The operators in the simple robot domain.

Table 1.

Consider a planning problem with the initial state as shown in the figure. Suppose that our goal is to bring the ball into room 3. This may be achieved by breaking through the wall of room 4 (**break**(1,4)) and then throwing the ball into room 3 (**throw**(4,3)). □

To compare the quality of different plans, we introduce the notion of the *cost* of a plan. We associate some positive cost with each operator, and define the cost of a plan as the sum of the costs of its operators.

Definition 2.1 *The cost of a plan* Π *equals the sum of the costs of its operators (*not *including the initial and goal states):* $\text{cost}(\Pi) = \sum_{\alpha \in \Pi} \text{cost}(\alpha)$. *An* optimal *solution of a planning problem is a plan with the lowest cost that solves the problem.*

For example, suppose we have to move the robot from room 1 to room 4, and we use the plan (**go**$(1,2)$, **go**$(2,3)$, **go**$(3,4)$) to solve this problem. The cost of this plan is $2+2+2=6$. Our solution is not optimal, since the same goal can be achieved by **break**$(1,4)$ with a cost of 4.

2.2 Primary-effect Justified Plans

If an operator α has several effects, we may choose certain "important" effects among them and use α only for the sake of these effects. The chosen important effects are called *primary* and denoted by *Prim-Eff*(α). The other effects are called *side* effects. For example, we can view the literal `door`(x,y) as a primary effect of the operator **break**(x,y), and `robot-in`(y) as a side effect.

A plan is *primary-effect justified* if it does not contain "useless" operators [Yang and Teneberg, 1990].

Definition 2.2 *Let* l *be a primary effect of an operator* α_1 *in some plan. We say that* l *is a* justified primary effect *if there is an operator* α *with a precondition* l *such that*

1. α_1 *is necessarily before* α*, and*
2. *there is no operator necessarily between* α_1 *and* α *that achieves or deletes* l

with either primary or side effects.

Informally this condition states that the precondition l of α is achieved by α_1 using a primary effect in some linearization of the plan. For example, consider the robot planning domain with the selection of primary effects given in the last subsection. The plan (**go**(1,2), **go**(2,3)) is primary-effect justified for achieving the goal `robot-in`(3). In contrast, the plan (**break**(1,3)) is not primary-effect justified, since changing the robot's position is *not* a primary effect of **break**.

2.3 Primary-effect Restricted Planners

Consider solving a planning problem defined by an initial state α_{init} and a goal α_{goal}. A partial-order planner starts with a two-operator plan $(\alpha_{\text{init}}, \alpha_{\text{goal}})$. It then inserts new operators and imposes ordering and variable binding constraints. While inserting a new operator, an *unrestricted* planner may use *any* operator that achieves l. In contrast, a *primary-effect restricted* planner always uses an operator with l as a *primary* effect.

We use a primary-effect restricted version of the TWEAK planner [Chapman, 1987], presented in Table 2, to illustrate the use of primary effects in the paper. Our techniques, however, are not limited to TWEAK; any backward-chaining planning system can benefit from the use of primary effects in the same way.

The version of TWEAK in Table 2 is called Prim-TWEAK. For simplicity, the algorithm description in Table 2 does not show how Prim-TWEAK treats the variables in preconditions and effects of operators; all variables are simply instantiated by constants. However, in our implementation of Prim-TWEAK, used in the experiments of Section 7, we do impose variable binding constraints.

Step 3A of the algorithm achieves a precondition l of α by ordering the operators in the plan, while Step 3B achieves l by inserting a new operator. Step 6 makes sure that there is no operator with an effect $\neg l$ between α_1 and α. The search for unsatisfied preconditions in steps 1 and 2 can be done in polynomial time (see the Modal Truth Criterion in [Chapman, 1987]).

We emphasize that step 3B is the only place where the algorithm is restricted to using primary effects. If *all* effects of all operators are selected as primary, then primary-effect restricted and unrestricted planners are equivalent. This would correspond to removing the word "primary" from step 3B of the algorithm and treating *all* effects as primary. In this case, the algorithm becomes identical to the original TWEAK planner.

Branching points in the description of the algorithm indicate places where the planner may consider different modifications of the current plan, thus creating several different branches of the search space. For completeness, the planner must consider *all* alternatives: it must try all possible operators on Step 3 and all possible orderings on Step 6.

To solve a planning problem, Prim-TWEAK is called with the initial plan $(\alpha_{\text{init}}, \alpha_{\text{goal}})$. If Prim-TWEAK cannot find a solution to the problem, it either terminates without any output or runs forever.

Example Consider again the robot domain, with the following selection of primary effects:

Prim-TWEAK(Π)
1. If all operator preconditions in the plan Π are necessarily satisfied, then return Π.
2. Choose some operator α with an unsatisfied precondition l.
3. Let α_1 be either
   (A) an operator of the plan Π possibly before α with an effect l, or
   (B) an operator in the library with a *primary* effect l.
   Branching point: each α_1 corresponds to a different branch in search space.
4. If α_1 is an operator from the library, add it to Π (without ordering constraints).
5. Order α_1 before α.
6. For every α_2 with an effect $\neg l$ possibly between α_1 and α
   (A) order α_2 before α_1, or
   (B) order α_2 after α, or
   (C) choose an operator with an effect l in Π and order it between α_2 and α.
   Branching point: different orderings correspond to different branches of the search space.
7. Let Π' be the resulting plan. Recursively call Prim-TWEAK(Π').

Table 2: Primary-effect restricted version of the TWEAK planner.

go(x,y)	{`robot-in`(y)}	‖	**throw**(x,y)	{`ball-in`(y)}
carry-ball(x,y)	{`ball-in`(y)}	‖	**break**(x,y)	{`door`(x,y)}

Assume that the initial state is as shown in Figure 2, and the goal is to place the robot in room 3. The goal may be achieved by breaking through the wall between rooms 1 and 3. However, changing the position of the robot is *not* a primary effect of breaking through a wall, and *Prim-Tweak* will not consider this possibility. Instead, it will find the solution plan (**go**(1,2), **go**(2,3)). □

3 Completeness and Cost Increase

Having introduced the concept of primary-effect restricted planners, we now present the first main result of the article. In this section, we point out two possible problems of planning with primary effects: losing completeness of planning and generating non-optimal plans. Then we present a necessary and sufficient condition of completeness of primary-effect restricted planning. This condition will enable us to construct an algorithm for learning primary effects (Section 5).

3.1 Completeness

One way to define a "good" planning algorithm is in terms of its ability to find a correct plan for every solvable planning problem. A planner that satisfies this property is said to be *complete*. For example, the unrestricted TWEAK algorithm is proved to be complete [Chapman, 1987].

When using primary effects, we would like every solvable problem to have a primary-effect justified solution to ensure completeness. A selection of primary effects with this property is

called a *complete* selection. If a goal cannot be achieved by a primary-effect justified plan, then a primary-effect restricted planner can fail to find a solution. We demonstrate this point through the robot planning example.

Consider the following selection of primary effects:

go(x,y)	{`robot-in`(y)}	‖	**throw**(x,y)	{`ball-in`(y)}
carry-ball(x,y)	{`ball-in`(y)}	‖	**break**(x,y)	{`door`(x,y)}

Assume that the initial state is as shown in Figure 2, and the goal is to move the robot out of room 1. The formal description of this goal is {¬`robot-in`(1)}. This may be achieved by the operator **go**$(1,2)$ or by **break**$(1,3)$. However a primary-effect restricted planner will fail to find either plan, because ¬`robot-in` is not a primary effect of *any* operator. To preserve completeness, we have to add more effects to our selection:

go(x,y)	{`robot-in`(y),¬`robot-in`(x)}
carry-ball(x,y)	{`ball-in`(y)}
throw(x,y)	{`ball-in`(y), ¬`ball-in`(x)}
break(x,y)	{`door`(x,y)}

Now the selection of primary effects is complete.

The completeness of planning also requires that a primary-effect restricted planner be able to find a solution of any problem that has a primary-effect justified solution. The Prim-TWEAK algorithm can be shown to satisfy this property.

Theorem 3.1 *If Prim-TWEAK expands nodes of the search space in the least-cost-first order, then it will find a solution plan for any problem that has a primary-effect justified solution.*

A proof of this theorem is similar to the completeness proof for unrestricted TWEAK (see, for example, [Yang and Murray, 1993]). In sum, planning with a set of primary effects is complete if (1) the set of primary effects is complete, and (2) the planner is primary-effect complete. In the rest of the paper, we will concentrate on the problem of finding a complete selection of primary effects for a given planning domain.

3.2 Solution Quality and the Cost Increase

Our next concern is the quality of solutions found by a planning algorithm. The unrestricted TWEAK planner is able to find optimal solutions for planning problems, but the use of primary effects may compromise this property of TWEAK and result in generating non-optimal plans. For example, consider our last selection of primary effects in the robot domain and suppose that initially the robot is in room 4. The optimal plan for traveling from room 4 to room 1 is a single operator **break**(4,1), the cost of which is 4. However, the cheapest solution found by a primary-effect restricted planner using the selected primary effects is (**go**(4,3), **go**(3,2), **go**(2,1)), with a cost of 6.

To formalize the decrease in solution quality due to the use of primary effects, we introduce the notion of the *cost increase*. Let Π be an optimal solution of a planning problem,

and Π' be a cheapest primary-effect justified solution. The ratio $\frac{cost(\Pi')}{cost(\Pi)}$ is called the *cost increase* for the planning problem. In our robot planning example, the cost increase is $6/4 = 1.5$. If planning with primary effects is complete and the cost increases of all planning problems have a finite upper bound, then the least upper bound of this set, C, is called the *greatest cost increase*. In other words, C is the smallest real number such that for *every* initial state α_{init}, *every* goal α_{goal}, and *every* unrestricted plan Π that achieves α_{goal} starting from the initial state α_{init}, there is a primary-effect justified solution plan with a cost at most $C \cdot cost(\Pi)$ that achieves the goal α_{goal} from the initial state α_{init}. Clearly, the greatest cost increase C is always greater than or equal to 1. The smaller the value of C, the better the selection of primary effects in terms of the quality of solution plans.

3.3 Necessary and Sufficient Condition of Completeness

In this section, we relate the value of the greatest cost increase to the completeness property of primary-effect restricted planning. Specifically, we present a theorem that allows us to test whether a given selection of primary effects is complete and to estimate the greatest cost increase.

Consider a plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ consisting of only one operator α, where α_{init} is an initial state satisfying the preconditions of α and the goal of the plan $\alpha_{\text{side-eff-goal}}$ is to achieve all the side effects of α while preserving all literals of the initial state not changed by α. The side effects of α may be formally expressed as $(Eff(\alpha) - Prim\text{-}Eff(\alpha))$, and the literals of the initial state α_{init} not changed by α are $(\alpha_{\text{init}} - Eff(\alpha))$. The formal expression for the goal is

$$\alpha_{\text{side-eff-goal}} = (\alpha_{\text{init}} - Eff(\alpha)) \cup (Eff(\alpha) - Prim\text{-}Eff(\alpha))$$

A *replacing plan* for $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ is a primary-effect justified plan that achieves the goal $\alpha_{\text{side-eff-goal}}$ from the same initial state α_{init}. In other words, a replacing plan must (1) achieve all side effects of α, and (2) leave all other literals of α_{init} unchanged.

As an example of a replacing plan, suppose that initially the robot is in room 4. Consider the plan with a single operator **break**(4,1). The side effects of this operator are `robot-in`(1) and ¬`robot-in`(4). The plan (**go**(4,3), **go**(3,2), **go**(2,1)) is a replacing plan, since it achieves both side effects of the **break** operator and does not change any other literals.

The *replacing cost increase* C_r of $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ is the ratio of the cost of an optimal replacing plan Π to the cost of α, $C_r = \frac{cost(\Pi)}{cost(\alpha)}$. In our example the cost of the operator **break**(4,1) is 4, and the cost of the replacing plan (**go**(4,3), **go**(3,2), **go**(2,1)) is 6, and thus the replacing cost increase is $\frac{6}{4} = 1.5$.

If the values of replacing cost increase of all one-operator plans have a finite upper bound, then their least upper bound is denoted by $C_{\max}$.

Theorem 3.2

Completeness: *Primary-effect restricted planning is complete* if and only if, *for every operator α and every initial state α_{init} satisfying the preconditions of α, the one-operator plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ has a replacing plan.*

Cost increase: *If replacing cost increases of one-operator plans have a finite maximum, $C_{\max}$, then the greatest cost increase for all solution plans of any sizes equals* $\max(1, C_{\max})$.

Informally, the validity of this theorem may be justified as follows. If we are given an unrestricted plan Π that solves some planning problem, then we may replace every operator of this solution by a corresponding replacing plan. The resulting plan is primary-effect justified and its cost is at most $C_{\max} \cdot cost(\Pi)$. A formal proof is presented in Appendix A.

Theoretically, the theorem can be applied to check whether a given choice of primary effects ensures the completeness and near-optimality of a primary-effect restricted planner, by examining each operator in the domain and finding a replacing plan for the operator's side effects in every possible situation. The planner is complete if replacing plans can be found for every operator. However, this approach is clearly impractical, since the number of operators and situations to be examined can be huge. Instead, we will use the theorem as a guideline to design a learning algorithm for automatically selecting primary effects of operators (Section 5).

The condition stated in the theorem is necessary for the completeness of planning when every set of literals could be a goal. However, it may be too conservative if, in practice, we encounter only some subclass of all possible problems. In this case, we could strengthen the theorem significantly by restricting the set of literals that replacing plans cannot affect. In Section 8.3 we discuss the notion of *problem-dependent* primary effects.

3.4 Redundant Primary Effects

Completeness is not the only quality measure for a selection of primary effects. Our next concern is avoiding redundancy when selecting primary effects.

Intuitively, a primary effect is redundant if it can be changed to a side effect without compromising completeness and without increasing the costs of solutions produced by a primary-effect restricted planner. Redundancy is undesirable: it increases the search complexity of planning without improving the quality of the solution plans.

Definition 3.1 *A selection of primary effects is called* redundant *if there exists a primary effect that can be changed to a side effect without increasing the greatest cost increase C.*

Let us look again at the example at the end of Section 2.3. Suppose that the **carry-ball**(x, y) operator has two primary effects: `robot-in`(y) and `ball-in`(y). In this case, "demoting" `robot-in` to the position of a side effect of **carry-ball** does not increase the costs of primary-effect justified solution plans, because moving the robot to a new place can always be accomplished by the cheaper operator **go**. Thus, making `robot-in` a side effect of **carry-ball** would improve the quality of the selection.

Avoiding redundancy will be one of our main concerns in selecting primary effects. In general, the task of finding the minimal non-redundant set of primary effects for a given cost increase is undecidable. However, in practice the problem can be solved approximately by applying learning techniques. In Section 5 we present a learning algorithm that in most cases avoids redundancy.

3.5 Summary

To help the reader understand the rest of the paper, we summarize the technical terms defined in this section:

Unrestricted Planner A planner that makes no distinction between primary and side effects of operators.

Primary-effect Restricted Planner A planner that inserts an operator into a plan only for achieving the primary effects of the operator.

Primary-effect Justified Plan A plan in which every operator has a justified primary effect, where a justified effect is an effect necessary for achieving a precondition of some other operator.

Primary-effect Complete Planner A primary-effect restricted planner that can solve every planning problem which has a primary-effect justified solution plan.

A Complete Selection of Primary Effects A selection for which every solvable problem has a primary-effect justified solution plan.

Cost of a Plan The sum of costs of all operators in the plan.

Cost Increase The ratio of the cost of the cheapest primary-effect justified solution plan to the cost of an optimal solution of a planning problem.

Greatest Cost Increase The greatest value of the cost increase among all planning problems.

Replacing Cost Increase The ratio of the cost of the cheapest primary-effect justified solution plan achieving the side-effects of an operator to the cost of the operator.

Redundant Primary Effect A primary effect that can be "demoted" to the position of a side effect without affecting the greatest cost increase.

4 Analysis of Search Reduction

In this section we compare the running time of a primary-effect restricted planner with that of an unrestricted planner. The goal of this comparison is to determine the conditions under which the use of primary effects reduces the running time of the algorithm. In the next section we show how to use the results of the analysis to guide a learning algorithm in its choice of primary effects.

We use the Prim-TWEAK and unrestricted TWEAK algorithms in our comparison. We assume that both planners use *best-first search*. That is, they always choose the cheapest incomplete plan to work on next. For simplicity of the analysis, we assume that all operators in our domain have the same cost. Thus, the cost of a plan is proportional to the number of operators in the plan.

To determine the time complexity of a planning algorithm, we analyze the search tree expanded by the planner. Each node in the planner's search tree corresponds to an incomplete plan considered by the planner during its search for a solution. *Expanding a node* corresponds to modifying a plan in this node by inserting a new operator (see step 3B in Table 2), imposing an ordering constraint (step 3A) and conflict resolution. As a result of

expanding a node, several successor plans may be generated in step 6. The branching factor B of search is the total number of successor plans generated after the execution of step 6.

The running time of a planner depends on the number of expanded nodes in the search tree and the time for processing each node. For TWEAK, the time spent on a single node is determined by the time of finding an unsatisfied precondition (step 2). This time is the same for both unrestricted and primary-effect restricted TWEAK. Thus, we use the number of nodes in the search space to compare the running time of Prim-TWEAK and that of unrestricted TWEAK.

We first consider unrestricted planning. We use B_{un} to denote the branching factor of unrestricted TWEAK. Assume that the size of an optimal solution to a planning problem is n. Then the total number of nodes in the search tree is

$$1 + B_{\text{un}} + {B_{\text{un}}}^2 + ... + {B_{\text{un}}}^n = \frac{{B_{\text{un}}}^{n+1} - 1}{B_{\text{un}} - 1} \tag{1}$$

Next consider the primary-effect restricted TWEAK planner with the greatest cost increase C. By definition of the cost increase, the number of operators in the solution found by Prim-TWEAK for the same problem is at most $C \cdot n$. Let the branching factor of Prim-TWEAK be B_{prim}. Then the number of expanded nodes is *at most*

$$\frac{{B_{\text{prim}}}^{C \cdot n+1} - 1}{B_{\text{prim}} - 1} \tag{2}$$

Let R denote the ratio of running times of primary-effect restricted and unrestricted planners. Since the running time spent per node is approximately the same for both planners, R is determined by the ratio of the search-tree sizes:

$$R \leq \frac{({B_{\text{prim}}}^{C \cdot n+1} - 1)/(B_{\text{prim}} - 1)}{({B_{\text{un}}}^{n+1} - 1)/(B_{\text{un}} - 1)} = O\left((\frac{{B_{\text{prim}}}^{C}}{B_{\text{un}}})^n\right) \tag{3}$$

This formula demonstrates the relation between the increase in the costs of plans, C, and the saving in the planning time, R. The use of primary effects reduces the planning time when $R < 1$.

We denote the base of the exponent in Formula 3 by r:

$$r = \frac{{B_{\text{prim}}}^{C}}{B_{\text{un}}} \tag{4}$$

Then Formula 3 may be rewritten as $R = O(r^n)$. If $r < 1$, the saving in running time grows exponentially with the size of the optimal solution, n. The smaller the value of r, the greater the saving.

Formula 4 shows that $r < 1$ when $\frac{{B_{\text{prim}}}^{C}}{B_{\text{un}}} < 1$. Solving this inequality w.r.t. C, we conclude that a primary-effect restricted planner performs better than an unrestricted planner when

$$C < \frac{\log(B_{\text{un}})}{\log(B_{\text{prim}})} \tag{5}$$

We use this formula in our learning algorithm in the next section.

From the above formulas we can make some general conclusions. First, if we reduce the number of primary effects, then the branching factor for primary-effect restricted planning B_{prim} should decrease. However, if too few effects are selected as primary effects, then it is possible that a primary-effect restricted planner has to apply more operators to achieve a goal, as compared to an unrestricted planner. This may lead to an increase in the C value. Thus, for a given selection of primary effects, whether or not a primary-effect restricted planner is more efficient than an unrestricted planner depends on how C compares with $\frac{\log(B_{\text{un}})}{\log(B_{\text{prim}})}$.

Second, Formula 4 demonstrates that it is always beneficial to avoid redundant primary effects. Recall that a primary effect is *redundant* if it can be demoted to a side effect without increasing C. Redundant primary effects increase the branching factor of primary-effect restricted planning, B_{prim}. Thus, if we could spot a redundant primary effect and change it to a side effect, we can only decrease the value of r. This leads to our conclusion that reducing the redundancy in a selection of primary effects always improves the efficiency of primary-effect restricted planning.

5 Automatically Finding Primary Effects

In this section we describe an inductive learning algorithm for automatically selecting primary effects of operators. The goal of the learning algorithm is to select primary effects that improve the efficiency of the planning algorithm, preserve completeness, and guarantee a high probability of finding a near-optimal solution. Our method is guided by the theoretical results of the last two sections. Theorem 3.2 of Section 3 states that for a primary-effect restricted planner to be complete and near-optimal within a cost increase C, there must be a replacing plan within this cost increase for every operator α_1 in every situation where α_1 is applicable. In Section 4 we demonstrated that a primary-effect restricted planner is more efficient than an unrestricted planner if $C < \frac{\log B_{\text{un}}}{\log B_{\text{prim}}}$. Here we use these two results to guide the design of our learning algorithm.

The procedure **Initial-Choice** (Table 3) performs a preliminary selection of primary effects. After this is done, the learning algorithm (Tables 4 to 5) is used to select additional primary effects in order to ensure the completeness and near-optimality.

5.1 Initial Choice

If some literal in a planning domain is not a primary effect of any operator, it cannot be achieved by a primary-effect restricted planner. Therefore, the completeness of planning requires that every achievable literal l be a primary effect of some operator. The algorithm for initial choice, presented in Table 3, makes sure that this condition is satisfied.

Initially, the algorithm asks the user to select primary effects of operators. All effects specified by the user are marked as primary and will remain primary in the final selection after the learning phase [1] (Section 5.2). If the user selects too few primary effects, or if the

[1] If the user selects too many primary effects, some of them may be redundant and could be changed back to side effects. In Section 8.2 we describe an algorithm for "de-selecting" effects to achieve this purpose.

Initial-Choice

1. For every operator α
   (A) ask the user to specify primary effects of α, and
   (B) make all user-selected effects primary.
2. For every achievable literal l that is not chosen by the user as a primary effect
   (A) find the cheapest operator α_{cheap} that achieves l, and
   (B) make l a primary effect of α_{cheap}.

Table 3: Finding an initial selection of primary effects.

Learner(UC, Δ, m)

1. **Loop**
2. For all operators α in the library, call **Learn-One-Operator**(α, UC, m).
3. Run Prim-Tweak and TWEAK to obtain average branching factors B_{prim} and B_{un}.
4. If $UC = 1$ or $UC < \frac{\log(B_{\text{un}})}{\log(B_{\text{prim}})}$, then **exit loop**.
5. $UC := \max(UC - \Delta, 1)$.
6. **End Loop**

Table 4: The Inductive Learner.

Learn-One-Operator(α, UC, m)

1. Set $counter = 0$.
2. **Loop**
3. Let α_{init} be a *random initial state* satisfying the preconditions of α.
   The process for generating the state is explained in detail in the text.
4. Define the goal state as $\alpha_{\text{side-eff-goal}} = (\alpha_{\text{init}} - \mathit{Eff}(\alpha)) \cup (\mathit{Eff}(\alpha) - \mathit{Prim\text{-}Eff}(\alpha))$.
5. Use Prim-Tweak to find a plan for the problem $(\alpha_{\text{init}}, \alpha_{\text{side-eff-goal}})$.
   The total cost of the plan must be no larger than $UC \cdot \mathit{cost}(\alpha)$.
6. If a plan within the cost limit $UC \cdot \mathit{cost}(\alpha)$ is found, then $counter := counter + 1$.
   Else, convert a side effect of α to primary, and reset $counter = 0$.
7. **exit loop** when $counter = m$.
8. **End Loop**

Table 5: Learning new primary effects for a single operator.

user does not select any primary effect at all, **Initial-Choice** and the learning algorithm will add the missing primary effects automatically.

The algorithm iterates through the literals in the domain. For every literal l that is *not* a primary effect of any operator in the user-specified selection, the algorithm finds a cheapest operator α_{cheap} that achieves l, and makes l a primary effect of α_{cheap}.

Example Suppose that the **Initial-Choice** algorithm is applied to our robot planning domain, and that the user has chosen `ball-in`(x) as the primary effect of the operator **carry-ball**(x, y). The procedure finds the cheapest operators achieving the remaining literals, `robot-in`, ¬`robot-in`, ¬`ball-in`, and `door`. The cheapest operator that achieves the literals `robot-in` and ¬`robot-in` is **go**, the cheapest operator for ¬`ball-in` is **throw**, and the cheapest operator for `door` is **break**. Thus, **Initial-Choice** selects the following primary effects:

go(x, y)	{`robot-in`(y), ¬`robot-in` (x)}	‖	**carry-ball**(x, y)	{`ball-in`(y)}
throw(x, y)	{¬`ball-in`(x)}	‖	**break**(x, y)	{`door`(x, y)}

5.2 Learning Additional Primary Effects

The procedure **Initial-Choice** may not select all primary effects needed to ensure the completeness property of the initial selection of primary effects. In this section, we discuss how additional primary effects are selected.

5.2.1 The Parameters UC, Δ and m

We start by explaining the meaning of the parameters used in the learning algorithm, **Learner**.

Recall that in Section 4 we have shown that a primary-effect restricted planner is more efficient than an unrestricted algorithm when $C < \frac{\log(B_{\text{un}})}{\log(B_{\text{prim}})}$. The goal of the **Learner** algorithm, presented in Table 4, is to find a selection of primary effects that satisfies this condition.

The **Learner** algorithm takes several parameters as its input. The first parameter, UC, is the initial cost-increase value. The final output of **Learner** will be a set of primary effects such that the resultant cost increase is no greater than UC and the condition $UC < \frac{\log(B_{\text{un}})}{\log(B_{\text{prim}})}$ is satisfied.

The **Learner** algorithm calls the **Learn-One-Operator** procedure, explained next, to select primary effects of each operator. After the algorithm finds a selection of primary effects for which the cost increase is at most UC, it checks whether $UC < \frac{\log(B_{\text{un}})}{\log(B_{\text{prim}})}$. The algorithm determines the branching factors B_{un} and B_{prim} for the current selection of primary effects either by performing experiments using a problem set supplied by the user, or by generating a number of random problems and accumulating the statistics when solving them. If the inequality for UC is satisfied, then primary-effect restricted planning is more efficient than unrestricted planning. In this case, the algorithm considers the selection successful and terminates.

If the inequality is *not* satisfied, however, the algorithm decrements UC by some "small" value Δ (where Δ is an input parameter) and calls the **Learn-One-Operator** procedure

for each operator again. This time the procedure selects additional primary effects to ensure a new cost increase, $(UC - \Delta)$. The reduction in the cost-increase value will have the effect that procedure **Learn-One-Operator** chooses more effects to be primary, which in turn reduces the depth of the search and the planning time. The algorithm continues to reduce UC until it finds a value of UC for which the use of primary effects is more efficient than unrestricted planning, or until UC drops to its smallest possible value, which is one.

The third input parameter, m, for the **Learner** algorithm is used in the termination condition of the **Learn-One-Operator** procedure (see Table 5). The value of m determines the probability of finding a primary-effect justified solution within the given cost increase in the future, when we use the resulting selection of primary effects in planning. In Section 6.1, we will derive the relationship between m and the probability of finding a solution.

If the user does not specify the input parameters of the **Learner** algorithm, the algorithm assigns default values to these parameters. In our implementation, the default values are $UC = 2$, $m = 10$, and $\Delta = 0.2$.

5.2.2 Learn From Examples

The core of our learning system is the **Learn-One-Operator** procedure, shown in Table 5. Given an operator α, this procedure generates a planning problem with a randomly selected initial state α_{init} and goal $\alpha_{\text{side-eff-goal}}$. The initial state is one that satisfies the preconditions of α. The goal state, on the other hand, is as described in Section 3.3: to achieve all side effects of α while leaving all other literals of the initial state unchanged. With the initial and goal states, the **Learn-One-Operator** procedure attempts to find a primary-effect justified solution to this planning problem using the primary effects selected so far. To ensure near-optimality, this solution must be within the cost increase UC.

If solutions can be found to m different problems of this type, with randomly generated initial states that satisfy the preconditions of α, **Learn-One-Operator** terminates. If, however, one of the random problems cannot be solved by a primary-effect justified solution within the cost increase UC, then a side effect of α will be chosen as a primary effect. Subsequently, the algorithm resets the *counter* variable and tests m more problems. Every time the learner fails to find a primary-effect restricted solution, it selects a new primary effect. Given that the total number of effects of α is E_α, the *counter* variable in the algorithm cannot be reset more than E_α times. Thus, for each operator, the learner will go through the loop no more than $E_\alpha \cdot m$ times.

5.2.3 Random Problem Generation

Step 3 of the **Learn-One-Operator** algorithm requires to generate a random initial state α_{init}. This state can be any problem state that satisfies the preconditions of α. Formally, let X_α be the set of all states that satisfy the preconditions of α, and let us consider a probability distribution μ_α on X_α, representing how often each state appears just before the operator α in linearly ordered solution plans. This is the distribution we wish to use in order to "train" the operator α in the learning process; that is, to learn primary effects of α.

There are two ways to obtain the distribution μ_α in practice. If we have a large number of previous solution plans saved in a plan library, we could use this library as a sample space

for approximating X_α and μ_α. This could be done by computing the relative frequency of the states that appear just before α in solutions from the plan library.

In the absence of a plan library, an alternative is to generate solution plans for problems with randomly generated initial and goal states, where random generation is based on the assumption that all states occur equally often, and then to extract from these solutions a distribution μ_α of the states in X_α. This represents a more conservative selection strategy; by testing every state with equal probability, we might select more primary effects than necessary. However, this might be the best thing we can do given the lack of additional domain knowledge.

Example Consider an application of **Learn-One-Operator** to our robot domain, with the initial selection taken from the previous example. Suppose that we call this algorithm to learn primary effects of the operator **throw**, and we have $UC = 1.5$. Consider the initial state α_{init}, where the robot and the ball are in room 1; this state satisfies the preconditions of **throw**(1,2). The operator **throw** does not have primary effects in the current selection, and its side effect is the new position of the ball. Next, the algorithm generates the goal state $\alpha_{\text{side-eff-goal}}$, where the goals are to move the ball into room 2 (which is the side effect of **throw**(1,2)) and to leave the robot in room 1 (which is the part of the initial state that must remain unchanged). Given the current selection of primary effects, the optimal primary-effect justified plan that achieves this goal is (**carry-ball**(1,2), **go**(2,1)). The cost of this plan is 5, while the cost of **throw**(1,2) is 2, and thus the cost increase is $5/2 = 2.5$, which is greater than the user-specified cost increase $UC = 1.5$. Therefore, the learner chooses the justified effect of **throw**, `ball-in`, as a new primary effect. (If the operator had several justified effects, the algorithm could choose any of them. However, in our example the operator **throw** has only one primary effect.) The selection of primary effects becomes as follows:

go(x,y)	{`robot-in`(y), ¬`robot-in`(x)}	‖	**carry-ball**(x,y)	{`ball-in`(y)}
throw(x,y)	{`ball-in`(y), ¬`ball-in`(x)}	‖	**break**(x,y)	{`door`(x,y)}

Now suppose that we call the **Learn-One-Operator** algorithm to learn additional primary effects of the operator **break**, and we still have $UC = 1.5$. Suppose that the algorithm generates an initial state α_{init} where the robot is in room 4 and the goal is to move the robot to room 1, which can be solved by applying the operator **break**(4,1), the cost of which is 4. (This plan is not primary-effect justified, since changing the location of the robot is *not* a primary effect of **break**.) The learner will find a primary-effect justified plan (**go**(4,3), **go**(3,2), **go**(2,1)) that achieves the same goal. The cost of this plan is 6, and thus the replacing cost increase is $6/4 = 1.5$, which is no greater than the user-defined cost increase $UC = 1.5$. Thus, the learner concludes that the effect `robot-in` of the operator **break** may be achieved by a primary-effect justified replacing plan, and does *not* choose it as a primary effect. □

5.2.4 A Heuristic Based on Operator Ordering

The **Learner** algorithm does not specify the order of processing operators. Different orders may result in different selections of primary effects. We wish to use an order that avoids

redundancy in selecting primary effects. Our experiments in several domains showed that if the algorithm processes operators in the order of increasing operator costs, it usually does not select redundant effects.

We now explain this heuristic in more detail. If some operator α_1 may be used in a replacing plan for another operator, α_2, then intuitively we should process α_1 first, since we can then use its newly selected primary effects while searching for a replacement of α_2. Because cheap operators are usually used in replacing more expensive ones, the heuristic favors processing operators in *ascending order* of operator costs. If α_1 and α_2 have the same cost, we may first process the operator with a smaller number of side effects. Intuitively, the larger the number of side effects, the higher the chance of choosing a primary effect incorrectly among them. If an operator with fewer side effects, say α_1, is considered first, we can use its newly selected primary effects when searching for a replacement of α_2, and thus reduce the number of candidates for a new primary effect among the side effects of α_2.

It could happen that although α_1 has few side effects, it has a large set of preconditions that are difficult to achieve. In this case, using α_1 in a replacing plan could increase the cost of a solution found by a primary-effect restricted planner. However, this case is already taken care of by the use of the cost increase value UC; if the replacing-plan cost is over $UC \cdot cost(\alpha_2)$, then more side effects of α_2 will be chosen as primary effects, and in doing so, the learning algorithm reduces the cost of solutions.

While in practice the above heuristic ordering almost always produces good results, it is *not* a guarantee against redundancy. In Section 8.2 we present an extension of the learning algorithm that is capable of detecting redundant effects and removing them from the selection.

The effectiveness of the learning algorithm depends on three main factors: its running time, the number of examples needed for generating a good selection of primary effects, and its power in reducing planning time. In the next section, we analyze the first two factors, and then, in Section 7, we demonstrate empirically that the learning algorithm can dramatically reduce the planning time.

6 Complexity Analysis of the Learning Algorithm

In this section, we derive a relationship between the value of m, used in the termination condition of the learning algorithm (see Table 5), and the probability that a near-optimal solution can be found using the selected set of primary effects. We also estimate the time complexity of the learning algorithm.

6.1 Failure Probability and the Termination Parameter m

We first analyze the "quality" of the primary effects selected by the **Learn-One-Operator** algorithm; that is, the probability that the selected primary effects are sufficient for finding primary-effect justified solutions to planning problems.

We consider learning primary effects for some operator α. For the set of primary effects of α selected by the **Learn-One-Operator** algorithm, we wish to determine the probability

that the operator *cannot* be successfully replaced by a primary-effect restricted plan within the cost increase UC.

Let μ_α be an arbitrary probability distribution defined on the set of states that satisfy the preconditions of the operator α. We draw, at random, a state α_{init} from the set of states that satisfy the preconditions of α, where the probability of drawing each state is determined by the distribution μ_α. Suppose that we try to find a replacing plan for α within the cost increase UC. We denote by $fail(\alpha)$ the probability that, for a randomly drawn initial state, such a replacing plan does *not* exist; we call this probability the *failure probability* of replacing α.

Let e_α be the number of effects of the operator α. The possible outcomes of the execution of the **Learn-One-Operator** algorithm for the operator α can be divided into the following $(e_\alpha + 1)$ cases:

Case$_0$: **zero** effects have been selected as primary effects;
Case$_1$: **one** effect has been selected as a primary effect;
Case$_2$: **two** effects have been selected as primary effects;
Case$_3$: **three** effects have been selected as primary effects;
...
Case$_{e_\alpha}$: all e_α effects have been selected as primary effects.

Let $fail_i$ be the failure probability in Case$_i$, and Prob(Case$_i$) be the probability that the execution of the **Learn-One-Operator** algorithm leads to Case$_i$. Then, the overall failure probability $fail(\alpha)$ is determined by the following equation:

$$fail(\alpha) = \Sigma_{i=0}^{e_\alpha} \text{Prob(Case}_i) \cdot fail_i \tag{6}$$

Since the algorithm terminates after m consecutive successful attempts to find a replacing plan for α, we have:

$$\text{Prob(Case}_i) \leq (1 - fail_i)^m$$

Substituting this bound for Prob(Case$_i$) into Formula 6, we get the following inequality:

$$fail(\alpha) \leq \Sigma_{i=0}^{e_\alpha} (1 - fail_i)^m \cdot fail_i \tag{7}$$

To find the maximal possible value of $fail(\alpha)$, we observe that, for $0 \leq x \leq 1$, the function $(1-x)^m \cdot x$ reaches its maximum at the point $x = \frac{1}{m+1}$, which can be verified by taking the derivative of this function and finding the points where the derivative is 0. Therefore, for $0 \leq x \leq 1$, we have:

$$(1-x)^m \cdot x \leq (1 - \frac{1}{m+1})^m \cdot \frac{1}{m+1} \leq \frac{1}{m+1}$$

Since, for every i, the value of $fail_i$ is between 0 and 1, we conclude from Formula 7 that

$$fail(\alpha) \leq \Sigma_{i=0}^{e_\alpha} \frac{1}{m+1} = \frac{e_\alpha + 1}{m+1} \tag{8}$$

Now suppose that the user wants to limit the failure probability by some small positive number δ; that is, we have to make sure that $fail(\alpha) \leq \delta$, for some user-specified value δ.

We can guarantee this bound on the failure probability by selecting a proper value of m: if $\frac{e_\alpha+1}{m+1} \leq \delta$, then $fail(\alpha) \leq \delta$. To satisfy the inequality $\frac{e_\alpha+1}{m+1} \leq \delta$, we set m as follows:

$$m = \left\lfloor \frac{e_\alpha + 1}{\delta} \right\rfloor \tag{9}$$

Note that if an optimal solution of some planning problem consists of n operators, then, for each operator of the solution plan, the probability of failing to replace this operator is δ and, therefore, the probability that we cannot replace at least one operator is bounded by $n \cdot \delta$. Thus, the probability that the problem does not have a primary-effect justified solution within the cost increase UC is bounded by $n \cdot \delta$.

Readers familiar with the computational learning theory may note the similarity between our analysis and PAC-learning methods. Indeed, our inductive learning algorithm was inspired by learning methods in the style of [Valiant, 1984]. A selection of primary effects corresponds to a *hypothesis* to be learned. In PAC learning, we search for a hypothesis that, with a high probability, is a good approximation of the target concept. A special case of PAC learning is finding the exact concept with high probability, and our learning algorithm corresponds to this special case: our algorithm finds a complete selection of primary effects with probability $(1 - \delta)$.

Following the style of the PAC-learning analysis, we now estimate the number of learning examples required for our algorithm. Since the algorithm takes at most m examples to learn each primary effect and an operator α may have at most e_α primary effects, the maximal number of examples that may be required to complete the learning process for an operator α is $e_\alpha \cdot m = O(\frac{e_\alpha^2}{\delta})$. If the planning domain contains k operators and e is the maximal number of effects of an operator, then the total number of examples for learning primary effects of all operators is determined by the following expression:

$$O(k\frac{e^2}{\delta})$$

This relationship between the number of examples and the reciprocal of the failure probability is called the *sample complexity* of a learning algorithm. For our algorithm, the failure probability δ can be made arbitrarily small by increasing the number of training examples.

6.2 Time Complexity

We now analyze the running time of the learning algorithm. Let L be the number of literals in the problem domain, and E be the total number of effects of all operators in the library, that is, $E = \sum_\alpha e_\alpha$. Then, the running time of the **Initial-Choice** algorithm is $O(L \cdot E)$.

The other part of the learning process is the execution of the **Learner** algorithm (Tables 4 and 5). The most expensive component of this algorithm is the application of the planning algorithm, Prim-TWEAK, to find a replacing plan. Let α_{cheap} be the cheapest operator in the library, α_{exp} be the most expensive operator, and UC be the greatest cost increase specified by the user. Then, the number of operators in a primary-effect justified replacement of a single operator is at most $a = UC\frac{cost(\alpha_{\text{exp}})}{cost(\alpha_{\text{cheap}})}$. Let p be the maximal number of preconditions of an operator in the problem domain and e be the maximal number of effects of an operator.

Then, the time spent for processing a single node in the search tree of Prim-TWEAK is $O(a^3e^2p)$ [Woods, 1991]. The number of nodes in the search tree is $O({B_{\rm un}}^a)$, where $B_{\rm un}$ is the branching factor of an unrestricted planner.

Finally, the number of examples considered by the learner is $O(k\frac{e^2}{\delta})$, where k is the number of the operators in the problem domain, and thus the Prim-TWEAK planner is invoked $O(k\frac{e^2}{\delta})$ times. Putting these expressions together, we find that the running time of learning is as follows:

$$O(\frac{a^3 \cdot e^4 \cdot p \cdot {B_{\rm un}}^a}{\delta} \cdot k)$$

where $a = UC\frac{cost(\alpha_{\rm exp})}{cost(\alpha_{\rm cheap})}$.

7 Planning Time Reduction

In this section, we present a series of experiments that demonstrate the effectiveness of our learning method in reducing the planning time in several different planning domains. In what follows, we first describe a family of domains used in our experiments. Then we show how the efficiency of primary-effect restricted planning in these domains depends on (1) the number of goals and the length of an optimal solution plan (2) the relative costs of operators, and (3) the number of different ways to achieve the same literal.

7.1 Artificial Test Domains

We have implemented Prim-TWEAK, unrestricted TWEAK, and the learning algorithm in Allegro Common Lisp on a Sun-4 Sparc Station. An option is added to the planner so that a problem can be solved either using Prim-TWEAK or unrestricted TWEAK. We call the combination of **Initial-Choice** and **Learner** algorithms **Prim-Learn**, which is implemented as a separate module and which calls TWEAK as a subroutine.

We ran our experiments with an artificial domain similar to the domains described in [Barrett and Weld, 1992]. The problem domain has a number of features that can be varied independently, enabling us to perform controlled experiments.

In this domain, a planning problem is defined by n initial-state literals, denoted by $init_1, init_2, .., init_n$, and n goal literals, $goal_1, goal_2, .., goal_n$. There are also n operators, named $Op_1, Op_2, .., Op_n$. Each operator Op_i has the single precondition $init_i$. Op_i removes the initial-state literal $init_{i-1}$ and establishes the goal literals $goal_i, goal_{i+1}, .., goal_{i+k-1}$. Thus, each operator has $(k+1)$ effects: one negative effect and k positive effects. (The goal literals are enumerated modulo n, that is, a more rigorous notation for literals established by Op_i is $goal_{i \bmod n}, goal_{(i+1) \bmod n}, .., goal_{(i+k-1) \bmod n}$.) The cost of the operator Op_i is denoted by $cost_i$.

In Lisp implementation, each operator Op_i is described as follows:

(def-operator
:name Op_i
:preconditions $init_i$
:effects ((not $init_{i-1}$) $goal_i$ $goal_{i+1}$ ... $goal_{i+k-1}$)

:cost $cost_i$

)

To help the reader understand the domain better, let us consider an example with $n = 10$ and $k = 1$. In this case, every operator Op_i achieves only one goal literal, and a solution plan is simply the sequence of operators:

$$Op_1, Op_2, Op_3, Op_4, Op_5, Op_6, Op_7, Op_8, Op_9, Op_{10}$$

On the other hand, if $k = 5$, then the first operator Op_1 achieves the first five goal literals, and Op_6 achieves the rest. Thus, a solution plan in this case is

$$Op_1, Op_6$$

Our artificial domain allows us to vary the following problem features:

Goal Size The number of goal literals, n, can be changed. The length of an optimal solution changes in proportion to the number of goal literals.

Cost Variation The *cost variation* is the statistical *coefficient of variation* of the costs of operators, defined as $(S_{cost}/\overline{cost})$, where S_{cost} is the standard deviation of the costs, and $\overline{cost}$ is their mean. Intuitively, the cost variation determines the relative difference between costs of different operators.

Effect Overlap The *effect overlap*, k, is the average number of operators establishing the same literal. In our experiments the effect overlap varies from 2 to 6.

Note that if the effect overlap is 1, then all effects must be selected as primary, and primary-effect restricted planning is equivalent to unrestricted planning.

We vary these three features in our controlled experiments. Even though this test domain is artificial, it demonstrates some important characteristics of many real-world domains. For example, in the real world if the goal size increases, one would expect that the solution plan for achieving the goals should also increase in size. Likewise, if the effect overlap increases then every operator can achieve a larger number of goals. As a result, the number of alternative solutions to a problem should increase, while each solution should get smaller. All of these phenomena could be shown to occur in our domain.

For each setting of the domain parameters, we run our learning algorithm for each planning operator to find primary effects of the operator. The termination parameter m is set to 10. The training set for each operator is generated using a random-state generator.

Our experiments show the following properties of planning with primary effects:

1. Planning with primary effects selected by **Prim-Learn** is, on average, more efficient than planning without primary effects.

2. The saving in planning time due to the use of primary effects exponentially increases with an increase of the solution length.

3. A significant reduction of planning time is achieved for any cost variation.

4. As the effect overlap increases, the efficiency improvement of planning with primary effects first increases and then decreases.

A small effect overlap results in smaller efficiency improvement because almost all effects are selected as primary, thus making primary-effect restricted planning close to unrestricted planning. A possible explanation of the efficiency decrease for a large overlap is that solutions found by the primary-effect restricted planner are longer than unrestricted solutions, which results in a larger search depth of planning.

7.2 Varying Solution Length

In this section we demonstrate that the saving in Prim-TWEAK's running time grows exponentially with the solution length. We consider domains with effect overlaps 2 and 4. The operator costs, $cost_i$, are either constant for all operators or linearly grow with i. The solution length of a problem is defined as the length of an *optimal* solution for this problem, found by unrestricted TWEAK. In all experiments, the initial user-defined cost increase UC is set to 2, the decrement Δ is set to 0.2, and m is set to 10. The number of operators and goal literals, n, varies from 1 to 20. Goals of the planning problems are random permutations of the goal set $\{goal_1, goal_2, .., goal_n\}$.

The results of experiments are presented in Figures 3 to 6. The graphs show the running times of unrestricted TWEAK (UT) and Prim-TWEAK (PT) for problems with different solution lengths. Every point on each figure is the average of five random problems solvable in 2 minutes of CPU time by unrestricted TWEAK.

For the effect overlap $k = 2$, we considered problems with an optimal solution up to 10 operators. Thus, for 20 goals an optimal solution contains only 10 operators. Likewise, for $k = 4$, we used problems with up to 5-operator solutions (because unrestricted TWEAK could not find longer solutions in 2 minutes). For every distinct set of operators, we ran the **Prim-Learn** algorithm to select primary effects. Then every problem was solved by both Prim-TWEAK and unrestricted TWEAK.

The running time taken by **Prim-Learn** is not shown in the graphs. Instead, it is summarized in Table 6, together with branching factors of both unrestricted TWEAK (UT) and Prim-TWEAK (PT). Recall that **Prim-Learn** is used only once for a given planning domain. If many problems are solved in the same domain, the amortized time for learning is usually negligible.

The figures show that the time saving of planning with primary effects increases with an increase of the solution length. Prim-TWEAK performs better than unrestricted TWEAK in all considered cases. Table 6 shows that the use of primary effects selected by **Prim-Learn** reduces the branching factor of search twofold.

7.3 Cost Variance

Next we consider the effect of cost variation on planning time. Each operator Op_i is associated with a cost value $cost_i$, $i = 1, \ldots, n$, which is a function of i. This function can be

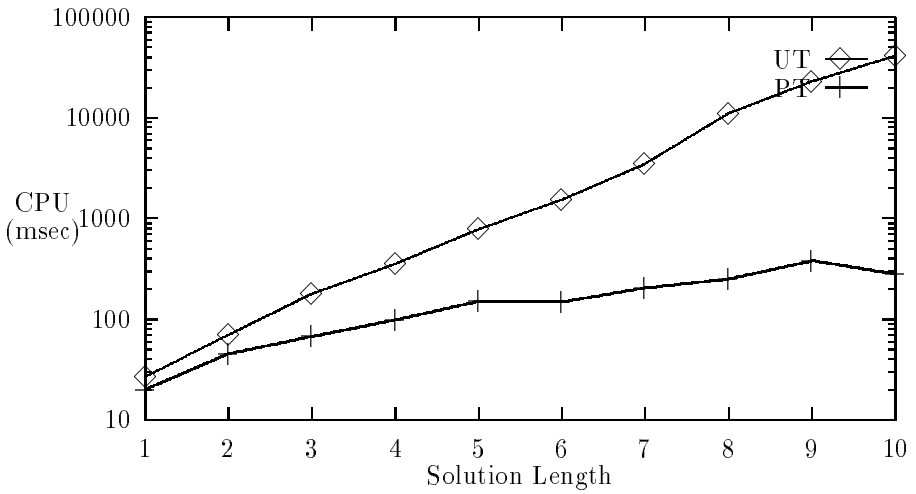


Figure 3: Effect overlap k equals 2, and all operators have the same cost.

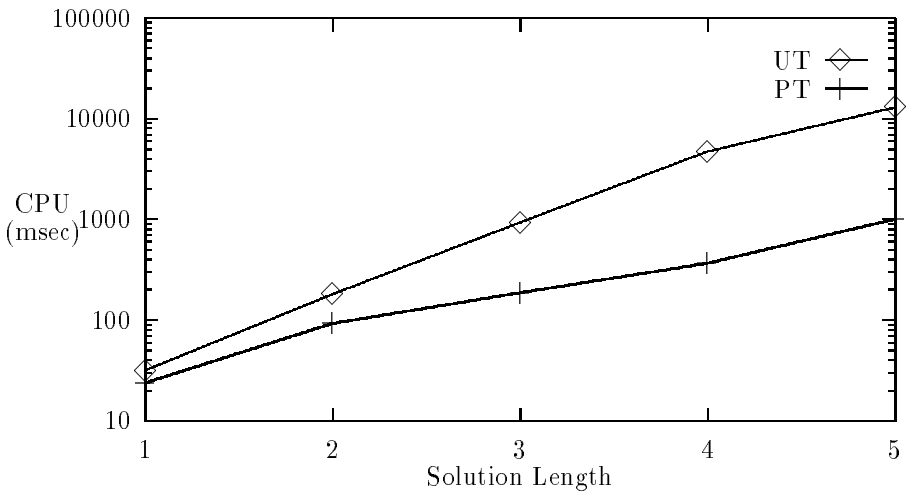


Figure 4: Effect overlap k equals 4, and all operators have the same cost.

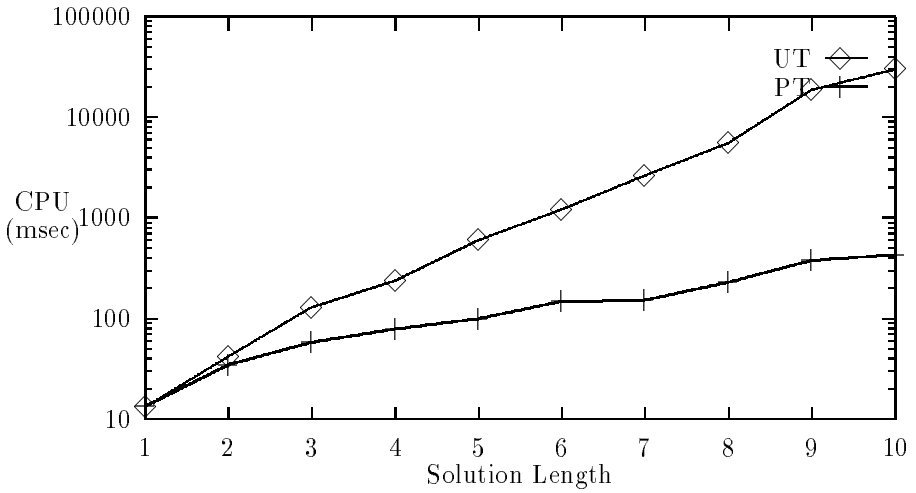


Figure 5: Effect overlap k equals 2, and the cost of Op_i linearly grows with i.

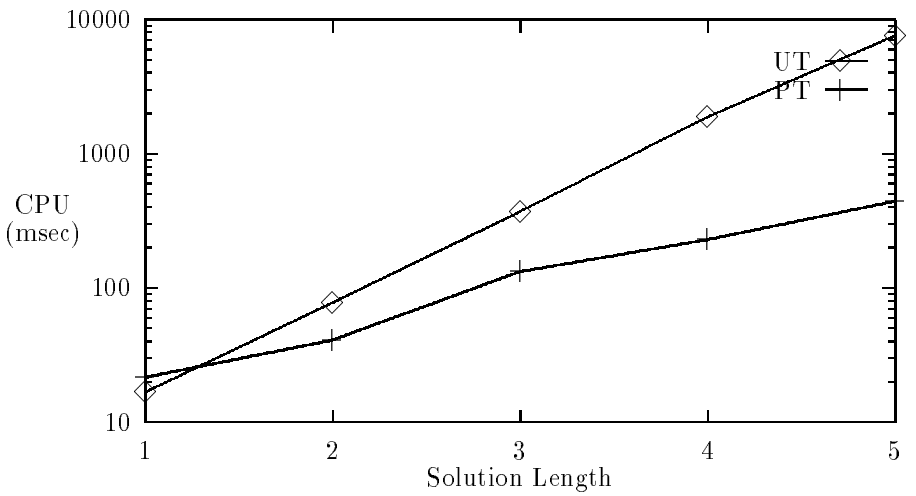


Figure 6: Effect overlap k equals 4, and the cost of Op_i linearly grows with i.

Overlap k	Learning Time (msec)		Average Branching Factor	
	UT	PT	UT	PT
2	0	3733	2.0	1.04
4	0	7533	4.0	1.61
6	0	12683	6.0	2.67

Table 6: Learning time and branching factor. (The number of operators is $n = 20$)

Cost function	Cost variation	PT / CPU (msec)	UT / CPU (msec)
Constant, $cost(Op_i) = 1$	0.0000	338.67	42910.84
Linear, $cost(Op_i) = 2 \cdot i + 10$	0.3720	269.17	16469.67
Exponential, $cost(Op_i) = 2^i$	2.3805	202.83	17274.83

Table 7: Comparison for different costs of operators.

constant, linear, or exponential. The number of operators and goals in these experiments is $n = 20$. For every cost function, we ran our planners on 5 different problems. The initial value of UC is 2, and the decrement Δ is 0.2.

The test results are shown in Table 7. The first column lists the cost functions. The second column presents the statistical variation for each cost function. The third and fourth columns compare the CPU time of Prim-TWEAK (PT) and unrestricted TWEAK (UT) under each cost function. The table shows that the use of primary effects reduces the planning time for every cost function. In these experiments, Prim-TWEAK runs 60 to 130 times faster than unrestricted TWEAK.

7.4 Effect Overlap

Now we consider performance of Prim-TWEAK for different values of the effect overlap. Recall that the effect overlap k is defined as the average number of operators achieving the same literal. The number of operators and goals in our experiments is again $n = 20$. We consider domains with constant, linear, and exponential cost functions. We vary the effect overlap k from 2 to 6. (Recall that if $k = 1$, then primary-effect restricted planning is equivalent to unrestricted planning.) For each cost function and each effect overlap we ran **Prim-Learn** to select primary effects and then solved 5 different problems using primary-effect restricted and unrestricted planners. Figures 7 to 9 show the running times of Prim-TWEAK (PT) and unrestricted TWEAK (UT) for every effect overlap under different cost functions. The graphs show that, for every effect overlap, the use of primary effects results in considerable saving of running time.

We note from these graphs that when the effect overlap k increases past a certain value (4 or 5), the planning time for both planning algorithms decreases. This phenomenon has an

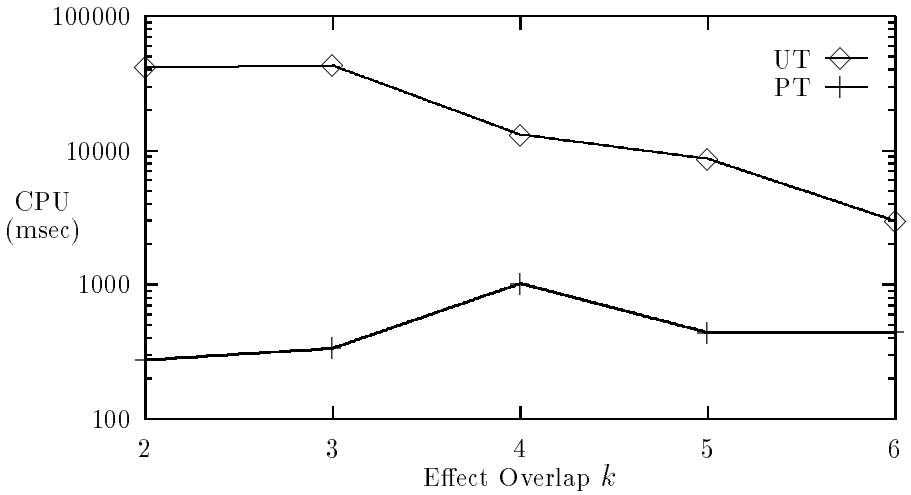


Figure 7: The cost function is constant: $cost(Op_i) = 1$.

explanation in terms of search complexity. With a fixed number of goals, when effect overlap increases, the number of goals that each operator can achieve also increases, which leads to the decrease in the length of an optimal solution. A decrease in the length of a solution implies a reduction in the depth of a planner's search. Therefore, for both Prim-Tweak and unrestricted Tweak, as k increases, the search depth decreases even though the branching factor increases. The net effect is that when k is greater than a certain value, the planning time decreases, as we observe from these graphs.

7.5 Experiments in a Robot Domain

In this section we describe the performance of our algorithm in a simple robot world, a modified version of the robot domain from [Fikes and Nilsson, 1971] (see Figure 10). The robot can move between rooms, open and close doors, carry boxes, and climb a table. The predicates in the domain include predicates describing locations and doors in each room, e.g., `location-inroom(door12, room1)`, predicates indicating the location of the robot, e.g., `robot-inroom(room3)`, predicates indicating the location of the boxes, e.g., `box-inroom(box1, room1)`, and predicates indicating the status of the doors, e.g., `status(door12, open)`. The actions of the robot are encoded by the operators listed in Tables 8 and 10. (The words preceded by "?" in the operator description denote variables in our implementation.) The full encoding of the table-climbing operators is shown in Figure 10.

We ran our learning algorithm with $\Delta = 0.2$, $m = 10$, and $UC = 2$. The learning time was 2.98 CPU seconds. The primary effects selected by the algorithm are shown in Table 8. Notice that these primary effects correspond to human intuition. For example, since pushing

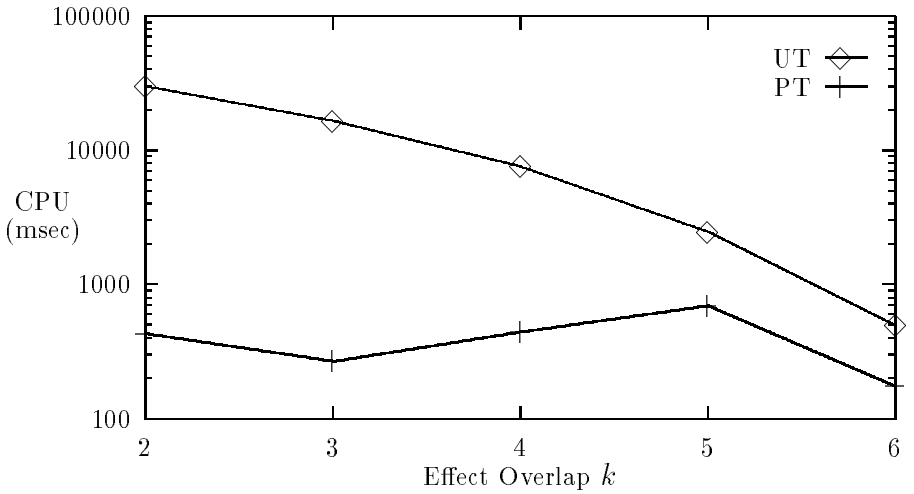


Figure 8: The cost function is linear: $cost(Op_i) = 2 \cdot i + 10$.

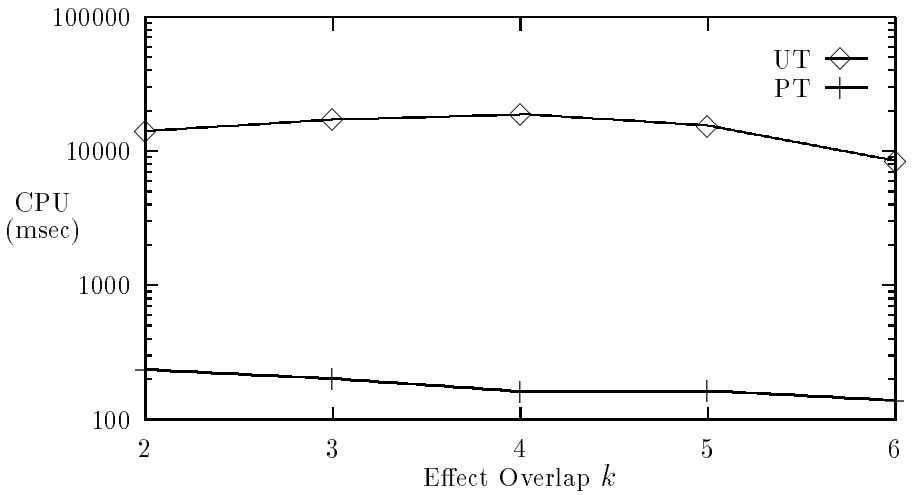


Figure 9: The cost function is exponential: $cost(Op_i) = 2^i$.

Effects	*Cost*	*Selected Primary Effects*
(goto-room-loc ?from ?to ?room)		
(robot-at ?to) (not robot-at ?from)	1	(robot-at ?to) (not robot-at ?from)
(go-thru-dr ?from-room ?to-room ?door)		
(robot-inroom ?to-room) (not robot-inroom ?from-room)	2	(robot-inroom ?to-room) (not robot-inroom ?from-room)
(climb-down ?table ?room)		
(robot-on-floor) (not robot-on ?table)	1	(robot-on-floor) (not robot-on ?table)
(open ?door ?room)		
(status ?door open) (not status ?door closed)	1	(status ?door open) (not status ?door closed)
(close ?door ?room)		
(status ?door closed) (not status ?door open)	1	(status ?door closed) (not status ?door open)
(push-box ?box ?from ?to ?room)		
(robot-at ?to) (box-at ?box ?to) (not robot-at ?from) (not box-at ?box ?from)	2	(box-at ?box ?to) (not box-at ?box ?from)
(push-box-thru-dr ?box ?from-room ?to-room ?door)		
(robot-inroom ?to-room) (box-inroom ?to-room) (not robot-inroom ?from-room) (not box-inroom ?from-room)	4	(box-inroom ?to-room) (not box-inroom ?from-room)
(carry-box-down ?box ?table ?room)		
(robot-on-floor) (box-on-floor ?box) (not robot-on ?table) (not box-on-table ?box ?table)	4	(box-on-floor ?box) (not box-on-table ?box ?table)

Table 8: The operators of the robot-box domain

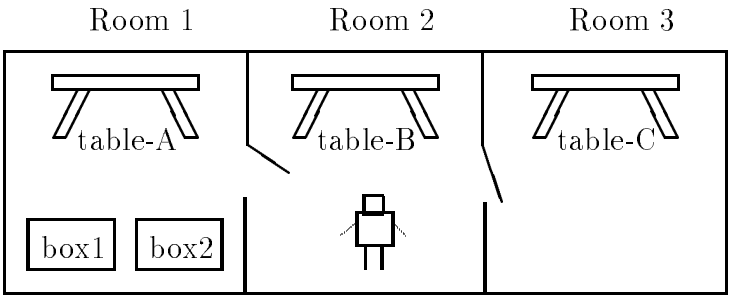


Figure 10: The robot domain used in the experiments

```
(defoperator                                  || (defoperator
  :name '(climb-up ?table ?room)              ||   :name '(carry-box-up ?box ?table ?room)
  :precond '((robot-at ?table)                ||   :precond '((robot-at ?table)
             (robot-inroom ?room)             ||              (robot-inroom ?room)
             (robot-on-floor)                 ||              (robot-on-floor)
             (table-inroom ?table ?room))     ||              (box-at ?table)
  :effects '((robot-on ?table) ;**            ||              (box-inroom ?box ?room)
             (not robot-on-floor)) ;**        ||              (box-on-floor ?box)
  :cost 1)                                    ||              (table-inroom ?table ?room))
                                              ||   :effects '((robot-on ?table)
                                              ||              (box-on-table ?box ?table) ;**
                                              ||              (not box-on-floor ?box) ;**
                                              ||              (not robot-on-floor))
                                              ||   :cost 4)
```

Table 9: Operators for climbing a table. Selected primary effects are marked by tt ";**".

Goal	Cost	Sol. Len	CPU time, ms		Branch. Factor		Expanded Nodes	
			PT	UT	PT	UT	PT	UT
(robot-on table-B)	1	1	50	50	1.0	1.5	2	2
(status door12 open)	2	2	183	217	1.4	2.0	7	7
(robot-inroom room1)	5	3	267	350	1.3	1.7	9	11
(box-at box1 door12)	7	5	800	1933	1.3	2.1	26	56
(robot-on table-C)	9	5	783	1200	1.3	2.3	26	36
(box-on box1 table-A)	7	5	800	1250	1.3	2.0	181	368
(status door12 closed) (robot-on table-A)	8	5	3466	12050	1.3	1.8	98	333
(box-at box2 table-B)	14	7	6550	14450	1.3	2.1	26	38

Table 10: Prim-TWEAK (PT) and unrestricted TWEAK (UT) in the robot domain

a box between rooms is more expensive than moving a robot alone, the algorithm selects the change in the position of the box as a primary effect of pushing a box.

Table 10 shows the performance of Prim-TWEAK (PT), with the primary effects selected by the learner, and unrestricted TWEAK (UT) on eight different problems in the robot domain. The initial state of all problems is as shown in Figure 10. All doors are initially closed, and the robot starts in room 2. In all eight problems, the Prim-TWEAK algorithm found optimal solution plans. (The costs of the optimal solutions are shown in the "Cost" column of Table 10, and the optimal solution lengths are shown in the next column.) As can be seen from the table, Prim-TWEAK is considerably more efficient in our robot domain than unrestricted TWEAK.

8 Extensions

We have developed a theory of primary-effect restricted planning and presented an algorithm for selecting primary effects. In this section we briefly mention several additional tricks that often improve the quality of selections of primary effects found by our algorithm.

8.1 Heuristics for Choosing Primary Effects

8.1.1 Improvements to Learn-One-Operator

The described algorithm **Learn-One-Operator** may be improved. When we choose a new primary effect among several justified effects of α, we may modify our algorithm to determine which effects of α are harder to achieve by a replacing plan. When finding a replacing plan, a primary-effect restricted planner can be modified to achieve as many justified effects of α as possible (but not necessarily all of them) by a replacing plan. This means that the planner inserts primary-effect justified operators in such a way that (1) their total cost is at most $UC \cdot cost(\alpha)$, (2) the preconditions of all newly inserted operators are satisfied, and (3) as many justified effects of α as possible are satisfied. Such a replacement may be found by Prim-TWEAK if it always achieves unsatisfied preconditions of newly inserted operators

De-selecting
For every operator α in the library:
1. Remove all primary effects of α added by **Learner**.
2. Call **Learn-One-Operator** with α.
3. If **Learn-One-Operator** selects a new primary effect for α during the last m examples, then go to step 2.

Table 11: De-selecting primary effects.

before achieving unsatisfied effects of α. If some of the justified effects of α are not achieved in the resulting plan, the learner selects one of these unachieved effects as a new primary effect.

8.1.2 Using Abstraction as a Selection Heuristic

Another heuristic in choosing primary effects is based on an algorithm for generating ordered abstraction hierarchies described in [Fink and Yang, 1992b]. This algorithm selects primary effects in such a way as to *maximize the number of levels in an abstraction hierarchy* for primary-effect restricted planning. Our experience shows that using this as a heuristic is often useful for finding good selections of primary effects. The algorithm helps avoid redundant primary effects and often leads to selections that correspond to the human intuition.

The use of this heuristic leads to the generation of a fine-grained ordered abstraction hierarchy for a resulting selection of primary effects. The use of this hierarchy in abstraction planning further improves efficiency [Knoblock, 1991a].

8.2 De-selecting Primary Effects

Once our learning algorithm has selected a primary effect, the effect remains primary and eventually becomes a part of the final selection. However, primary effects selected later can make this effect redundant. Besides, some primary effects may become redundant if we extend the library of operators in a problem domain with new operators or if we decide to set a higher cost increase *UC*. To avoid the problem of redundancy, the learning algorithm can be augmented to *de-select* some primary effects, that is, to convert them back into side effects.

The algorithm for de-selecting primary effects is presented in Table 11. Before learning primary effects of an operator α, the **De-selecting** algorithm *removes all primary effects of α learned previously.* When learning primary effects of α again, the algorithm will not select the redundant effects.

Upon execution, superfluous effects are removed by this algorithm from the current selection. However, some non-redundant primary effects may be removed in this process as well. To restore the completeness of the selection, we run the **Learner** algorithm again.

8.3 Problem-Dependent Primary Effects

Our theory of primary effects and the learning method can be considered as *problem-independent*, since they assume that every problem can be encountered in a given domain. In reality, we might be only interested in a subset of planning problems, and in that case, we could strengthen our results.

A subset of planning problems may be solved using only a subset O of the operators in a given domain. These operators in turn restrict the set L of literals that occur in solution plans. As a consequence, in Theorem 3.2, we only need to require that our replacing plans do not modify the literals in L, and we only need to worry about learning primary effects for operators in O. In addition, the learning algorithm needs to consider a smaller set of example plans. Thus the consequence of restricting our attention to a subset of problems is that the learning process will be more efficient, the selected primary effects will be fewer, and the primary-effect restricted planners will require less time to find solutions.

9 Conclusions

In this paper we have presented a formalism of the use of primary effects in planning and a learning algorithm for automatically selecting primary effects. Our main results are summarized below.

1. The use of primary effects may result in an exponential amount of reduction in planning time. However, an improper selection of primary effects can make a planner produce non-optimal solution plans, increase running time of a planner, or make a planner incomplete. To address this problem, we have identified several main factors that determine the completeness of planning with primary effects and presented a necessary and sufficient condition of completeness. The most important factor is the greatest cost increase C associated with a selection of primary effects. This factor not only determines the optimality of solutions found by a primary-effect restricted planner, but also the running time of the planner. If the greatest cost increase equals 1, then primary-effect restricted planning never performs worse than unrestricted planning.

2. We have presented an inductive learning algorithm that *automatically* selects primary effects. The algorithm finds a complete selection of primary effects by analyzing a set of example plans. The sample complexity of the algorithm is linear in the size of the library of operators, such that with a sufficient number of example plans, primary-effect restricted planning can produce near-optimal plans with high probability. Furthermore, our experimental results have shown that in most cases the learned primary effects help reduce the planning time exponentially.

Acknowledgement

The authors would like to thank Steve Minton and an anonymous referee for valuable comments.

References

[Anthony and Biggs, 1992] Martin Anthony and Norman Biggs. Computational Learning Theory. *Cambridge University Press*, Cambridge, UK, 1992.

[Bacchus and Yang, 1992] Fahiem Bacchus and Qiang Yang. The expected value of hierarchical problem-solving. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, 1992.

[Barrett and Weld, 1992] Anthony Barrett and Dan Weld. Partial order planning: evaluating possible efficiency gains. University of Washington, Department of Computer Science and Engineering, 1992. Tech. Report 92-05-01.

[Carbonell *et al.*, 1991] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. PRODIGY: an integrated architecture for planning and learning. Research Report, School of Computer Science, Carnegie Mellon University, 1991. Tech. Report CMU-CS-CS-89-189.

[Chapman, 1987] David Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32, pages 333-377, 1987.

[Fikes and Nilsson, 1971] Richard E. Fikes and Nils J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2, pages 189–208, 1971.

[Fink and Yang, 1992a] Eugene Fink and Qiang Yang. Formalizing plan justifications. *Proceedings of the Ninth Conference of the Canadian Society for Computational Studies of Intelligence (CSCSI)*, pages 9–14, 1992.

[Fink and Yang, 1992b] Eugene Fink and Qiang Yang. Automatically abstracting effects of operators. *Proceedings of the First International Conference on AI Planning Systems*, pages 243–251, 1992.

[Knoblock, 1991a] Craig A. Knoblock. *Automatically Generating Abstractions for Problem Solving.* Ph.D. Thesis, School of Computer Science, Carnegie Mellon University, 1991. Tech. Report CMU-CS-91-120.

[Knoblock, 1991b] Craig A. Knoblock. Search reduction in hierarchical problem solving. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 686–691, 1991.

[Knoblock *et al.*, 1991] Craig A. Knoblock, Josh D. Tenenberg, and Qiang Yang. Characterizing abstraction hierarchies for planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 692–697, 1991.

[Knoblock and Yang, 1993] Craig A. Knoblock, and Qiang Yang. A Comparison of the SNLP and TWEAK Planning Algorithms. In *Working Notes of the AAAI Spring Symposium Series: Foundations of Automatic Planning: The Classical Approach and Beyond*, pages 73-77, 1993.

[Korf, 1987] Richard E. Korf. Planning as search: a quantitative approach. *Artificial Intelligence*, 33(1), pages 65–88, 1987.

[Minton *et al.*, 1991] Steven Minton, John Bresina, and Mark Drummond. Commitment strategies in planning: a comparative analysis. *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 259–261, 1991.

[Minton *et al.*, 1992] Steven Minton, Mark Drummond, John Bresina, and Andrew Philips. Total order vs. partial order planning: factors influencing performance. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 83–92, 1992.

[Nilsson, 1980] Nils J. Nilsson. *Principles of artificial intelligence*. Tioga Pub. Co., Palo Alto, CA, 1980.

[Tate, 1977] Austin Tate. Generating project networks. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 888-893, 1977.

[Valiant, 1984] L.G. Valiant. A theory of the learnable. In *Communications of the ACM*, Vol. 27, pp. 1134-1142, 1984.

[Wilkins, 1988] David Wilkins. *Practical planning: extending the classical AI planning paradigm*. Morgan Kaufmann, CA, 1988.

[Woods, 1991] Steven G. Woods. *An implementation and evaluation of a hierarchical nonlinear planner*. Master's thesis, University of Waterloo, Department of Computer Science, Waterloo, Ont., Canada, 1991.

[Yang and Murray, 1993] Qiang Yang and Cheryl Murray. An evaluation of the temporal coherence heuristic for partial-order planning. To appear in *Computational Intelligence Journal* 1993.

[Yang and Tenenberg, 1990] Qiang Yang and Josh Tenenberg. ABTWEAK: abstracting a nonlinear, least commitment planner. In *Proceedings of Eighth National Conference on Artificial Intelligence*, pages 923–928, Boston, MA, 1990.

[Yang, Tenenberg and Woods, 1991] Yang, Qiang and Tenenberg, Josh and Woods, Steve. Abstraction in Nonlinear Planning. Technical report, CS 91-65, 1991, University of Waterloo, Department of Computer Science, Waterloo, Ontario, Canada N2L 3G1, 50 pages.

A Proof of Theorem 3.2

Completeness: *Primary-effect restricted planning is complete* if and only if, *for every operator* α *and every initial state* α_{init} *satisfying the preconditions of* α, *the one-operator plan* $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ *has a replacing plan.*

Cost increase: *Let* $C_{\max}$ *be the maximum of replacing cost increases. Then the greatest cost increase equals* $\max(1, C_{\max})$.

Proof We prove the second part of the theorem, that the greatest cost increase is equal to the maximum of replacing cost increases. The proof of the first part is similar.

Let $C_{\max}$ be the maximum of replacing cost increases. First assume that $C_{\max} \geq 1$. We have to show that

(1) the cost increase for every planning problem is no greater than $C_{\max}$, and

(2) there is a planning problem for which the cost increase is exactly $C_{\max}$.

(1) Consider an arbitrary planning problem with an optimal linear solution $\Pi = (\alpha_{\text{init}}, \alpha_1, \alpha_2 ..., \alpha_n, \alpha_{\text{goal}})$. We have to find a primary-effect justified plan achieving the same goal with a cost at most $C_{\max} \cdot \mathit{cost}(\Pi)$. We may convert our optimal solution into a primary-effect justified plan by replacing some of its operators. We begin by considering the last operator, α_n. If α_n is *not* primary-effect justified, we substitute a replacing plan instead of α_n. The cost of this replacing plan is at most $C_{\max} \cdot \mathit{cost}(\alpha_n)$. If α_{n-1} is *not* primary-effect justified in the resulting plan, we also substitute a replacing plan instead of it. Then we repeat this for α_{n-2}, α_{n-3}, etc., until all operators without justified primary effects are replaced by primary-effect justified plans. It may be shown that when we replace some operator α_i, all operators *after* it remain primary-effect justified. Therefore, we have obtained a *primary-effect justified* plan. The cost of this new plan is *at most*

$$C_{\max} \cdot \mathit{cost}(\alpha_1) + C_{\max} \cdot \mathit{cost}(\alpha_2) + ... + C_{\max} \cdot \mathit{cost}(\alpha_n) = C_{\max} \cdot \mathit{cost}(\Pi)$$

(2) Let $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ be such a single-operator plan that the replacing cost increase of $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ is $C_{\max}$, the largest of the replacing cost increases. Notice that primary effects of α are not goal literals. Thus, the plan $(\alpha_{\text{init}}, \alpha, \alpha_{\text{side-eff-goal}})$ is not primary-effect justified. The cost of this plan is $\mathit{cost}(\alpha)$, while the cost of the cheapest primary-effect justified plan that solves this planning problem is $C_{\max} \cdot \mathit{cost}(\alpha)$. Thus, the cost increase for this problem is at least $C_{\max}$.

Let us also consider the case when $C_{\max} < 1$. Then Part (1) of the proof shows that, for an unrestricted plan Π, each operator α of Π that is not primary-effect justified may be replaced by a subplan *cheaper* than α itself, and thus the cost of a new plan after all replacements is *at most* the cost of the initial plan Π. Thus, the cost increase cannot be larger than 1. On the other hand, the cost increase is *at least* 1. Thus, the cost increase for every problem is 1. □